My Understanding of "GPU Accelerated Static Timing Analysis"

Tau Workshop — Timing Conference (Tau 2021)

Zizheng Guo, Tsung-Wei Huang and Yibo Lin

2021-07-23

Presenter: Sethupathi Balakrishnan

The Electronic Design Automation Laboratory
Graduate Institute of Electronics Engineering
National Taiwan University
Taipei 106, Taiwan

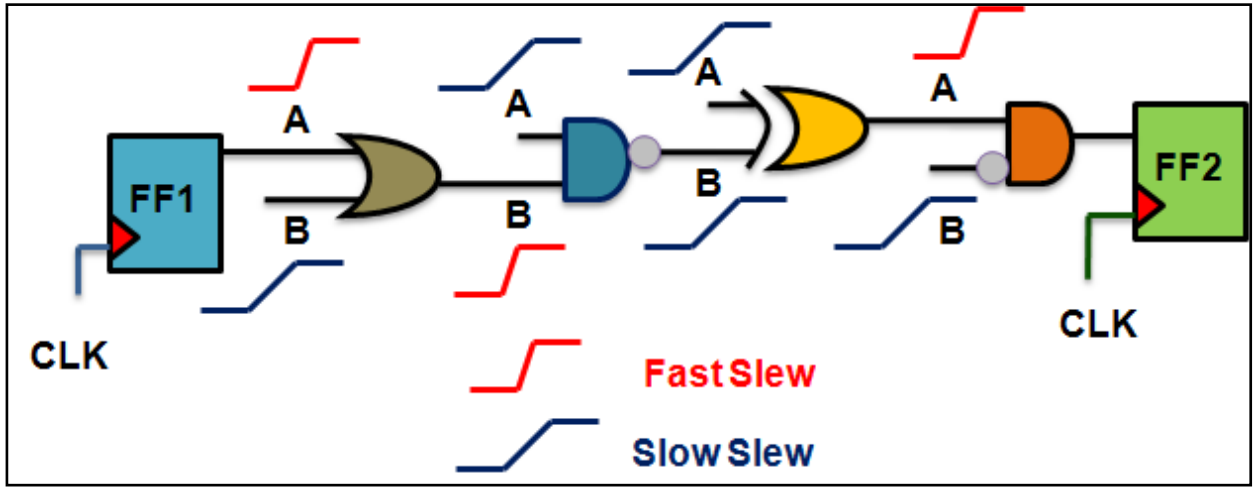
臺灣大學

Presentation Outline

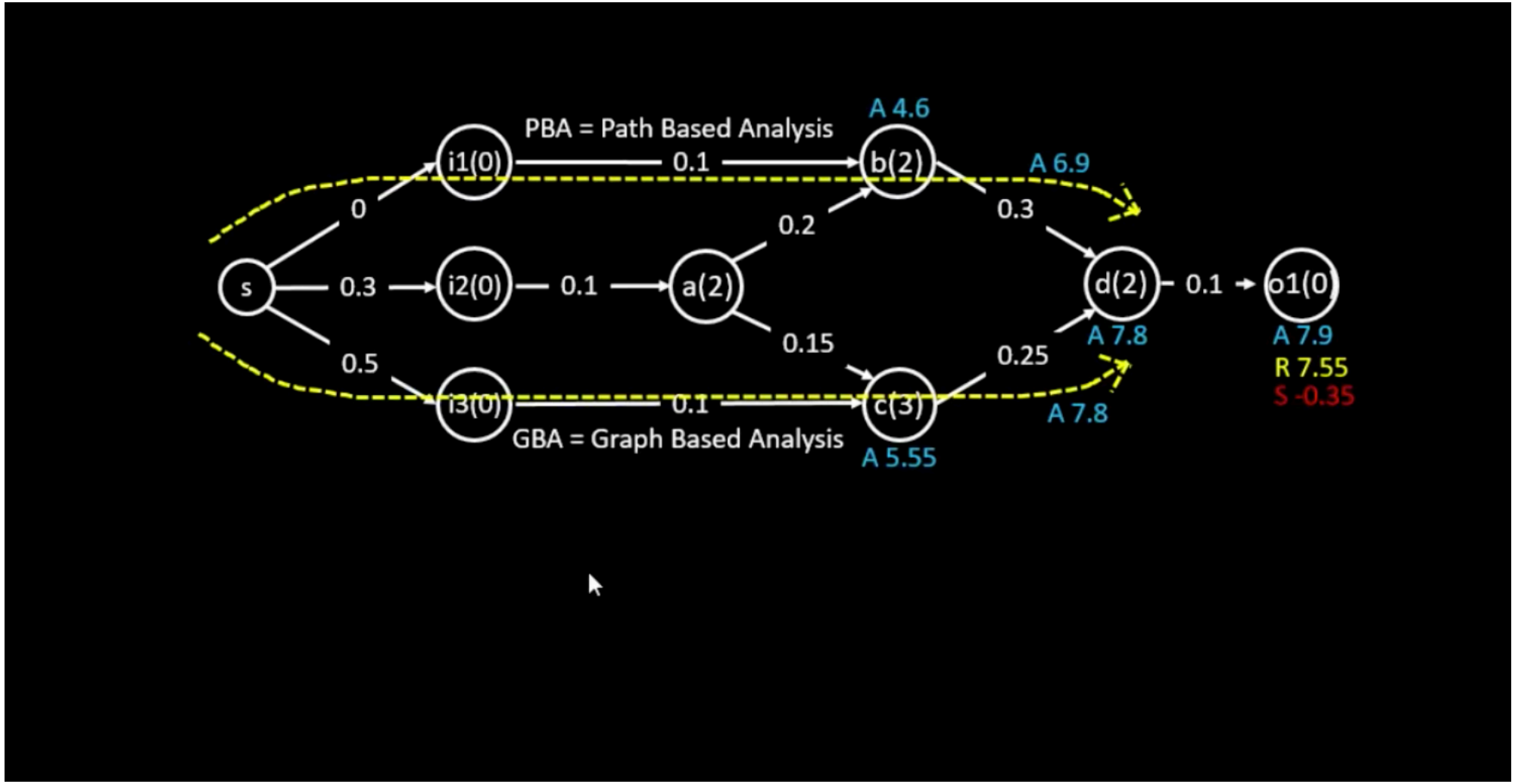
- **GBA vs PBA (An Introduction)**
- **CPU vs GPU**
- **GPU Accelerated STA**
- **RC Delay Computation**
- **NLDM based Cell delay Interpolation**
- **GPU Memory Access Latency Optimization**
- **Levelization**
- **Frontier/Advanced Frontier**

Core Idea (GBA vs PBA)

- We have 2 slews — fast and slow.
- In **Graph Based Analysis**, the timing engine computes the worst case delays of all standard cells assuming the worst case slew for all inputs of a gate. i.e slow slew for setup and fast slew for hold. GBA is faster because the engine has to simply report the worst case delay values at each cell and it is pessimistic. (The run time of the prime-time tool is less)
- In **Path Based Analysis**, the tool takes into account the actual slew for the arcs encountered while traversing any particular timing path. This way, the extra pessimism in delay calculation is avoided. However, this is achieved at a run-time cost. PBA calculation is slow because the tool has to trace the specific path and calculate accurate delays by taking into consideration the accurate slew values in the path. This is time consuming since there are usually millions of paths in a design.
- This paper uses GPU acceleration to speed up the GBA & PBA based delay calculation.



Example


PBA = Path Based Analysis
A 4.6
i1(0)
0.1
b(2)
A 6.9
0
0.2
0.3
s
0.3
i2(0)
0.1
a(2)
d(2)
0.1
o1(0)
A 7.8
A 7.9
R 7.55
S -0.35
0.15
0.25
0.5
i3(0)
0.1
c(3)
A 7.8
GBA = Graph Based Analysis
A 5.55

Introduction

- During the timing closure, **static timing analysis (STA) is frequently called in an inner loop of an optimization** algorithm to iteratively and incrementally improve the timing of the design
- Optimization engines apply millions of design transforms to modify the design and the timer has to quickly and accurately update the timing to ensure slack integrity.
- STA engines architected so far have been constrained by the multithreaded paradigm on a many core CPU platform. While the results show some scalability, most of them are not scalable beyond 8-16 threads.
- **Computing a timing graph involves irregular memory access and significant diverse computational patterns**, including graph-oriented computing, dynamic data structures, branch-and-bound, and recursion
-

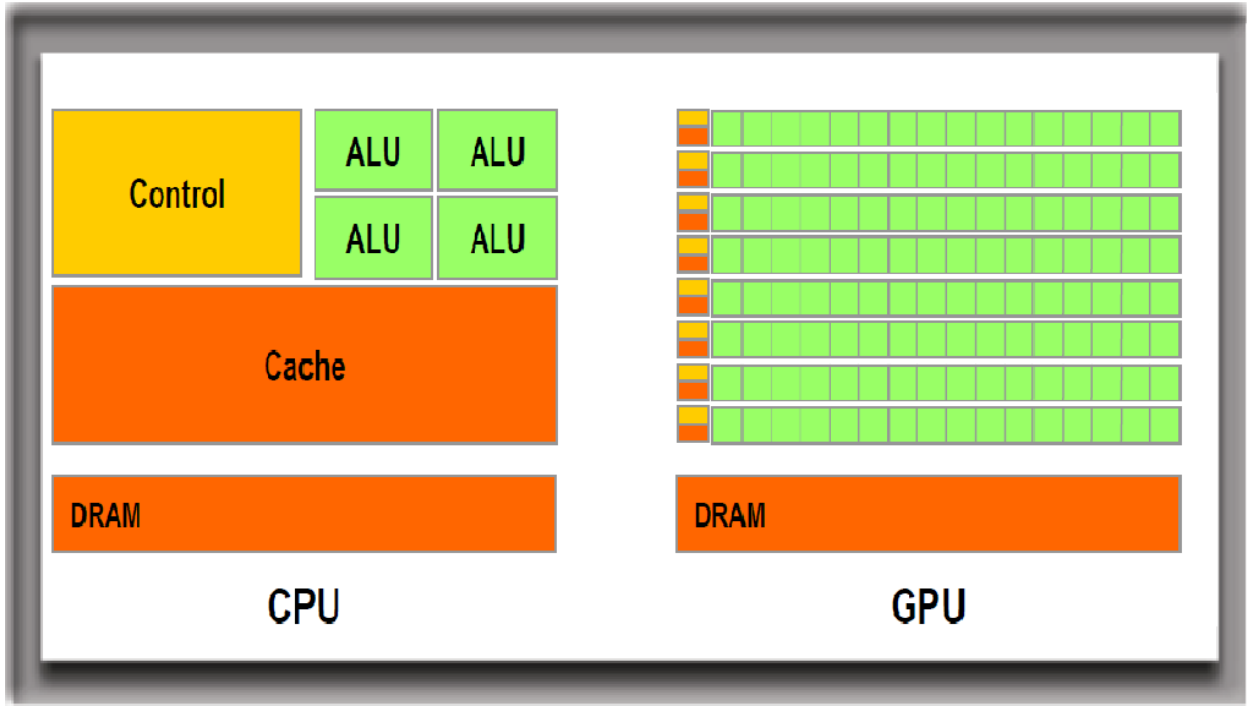
Contributions by the Authors

- They've leveraged task parallelism to decompose the STA workload into CPU-GPU dependent tasks and enable efficient overlap between data processing and kernel computation.
- They've developed GPU-efficient data structures and algorithms for delay computation, levelization, and timing propagation tasks, all of which are essential to update a STA graph.
- They've developed their GPU acceleration algorithms on top of a real- world STA engine that supports incremental timing on industrial design formats. Their techniques reflect realistic performance tradeoff between CPU and GPU.

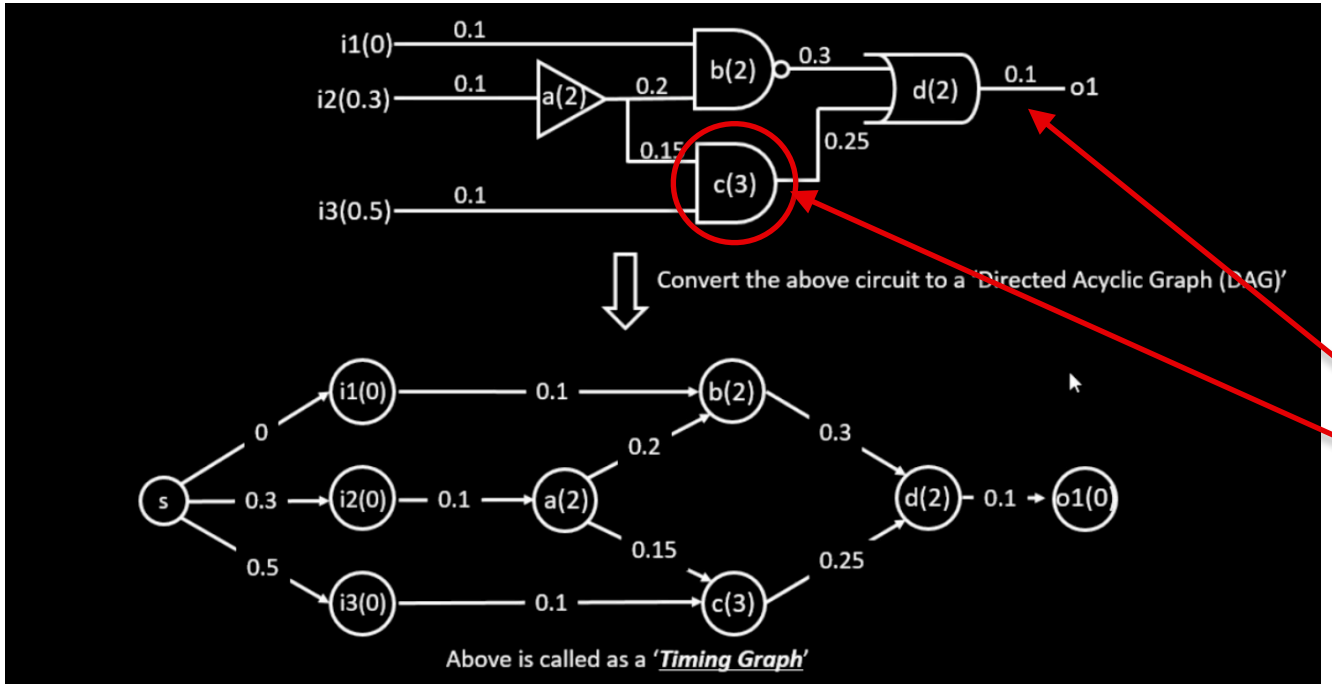
- **EXPERIMENTAL RESULTS —**
- They have evaluated their algorithm on industrial designs released by TAU 2015 Timing Analysis Contest. As an example, they've accelerated OpenTimer by **3.69X and 3.60X** on two large designs, leon2 (23M nodes and 25M edges) and netcard (21M nodes and 23M edges), using a single GPU. In the extreme, their implementation using one GPU can run faster than OpenTimer of 40 CPUs.
-

CPU vs GPU — Difference in Architecture

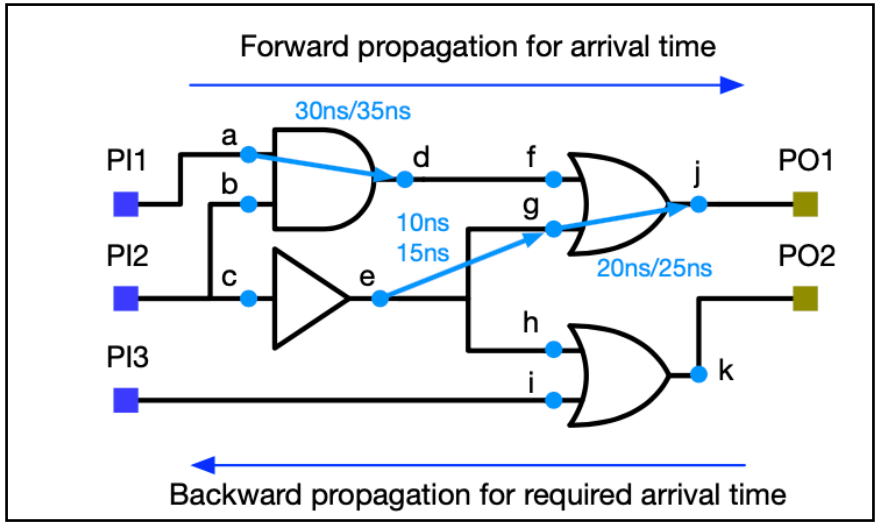
- The main difference between CPU and GPU architecture is that a **CPU is designed to handle a wide-range of tasks quickly** (as measured by CPU clock speed), but are limited in the concurrency of tasks that can be running. A GPU is designed to quickly render high-resolution images and video concurrently.
- Because **GPUs can perform parallel operations on multiple sets of data**, they are also commonly used for non-graphical tasks such as machine learning and scientific computation. Designed with thousands of processor cores running simultaneously, GPUs enable massive parallelism where each core is focused on making efficient calculations.
- While GPUs can process data several orders of magnitude faster than a CPU due to massive parallelism, GPUs are not as versatile as CPUs. CPUs have large and broad instruction sets, managing every input and output of a computer, which a GPU cannot do. In a server environment, there might be 24 to 48 very fast CPU cores. Adding 4 to 8 GPUs to this same server can provide as many as 40,000 additional cores.



STA — Timing Graphs



Modeling/Mapping Information	
Circuit	**Graph**
Wires	Edges
Gates	Nodes
I/O Ports	Nodes
RC Trees (Net Delay)	Elmore Delay Model
Gate Delays	NLDM 2D Grid Model



Note — It is somewhat easy to compute the RC net delay.
It is time consuming to compute Cell delay because

cell_delay = f(input transition, output capacitance)

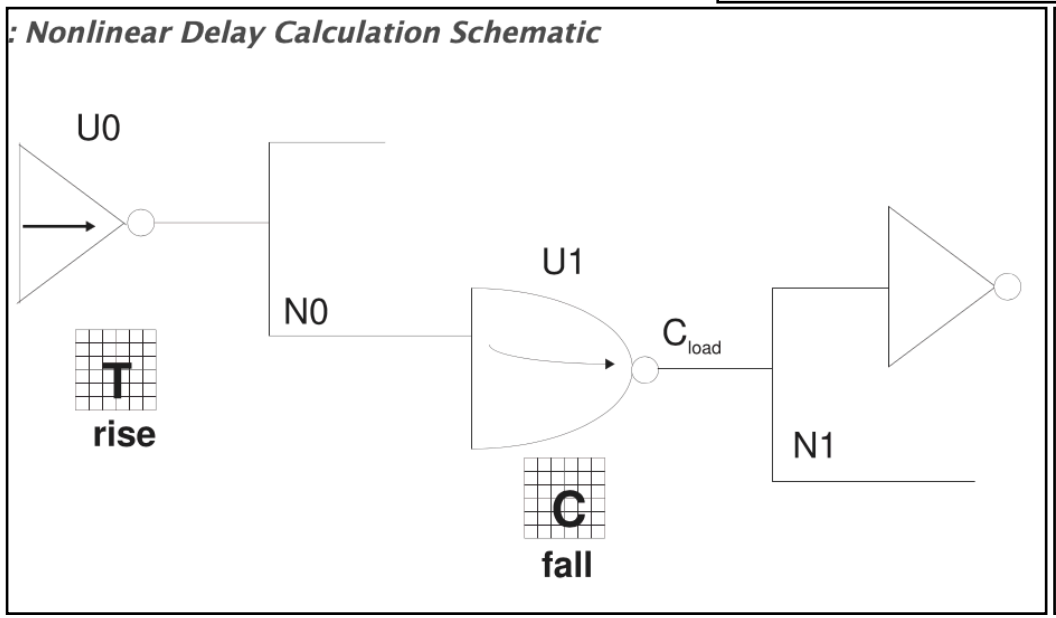
NLDM Delay Calculation (2D Grid)

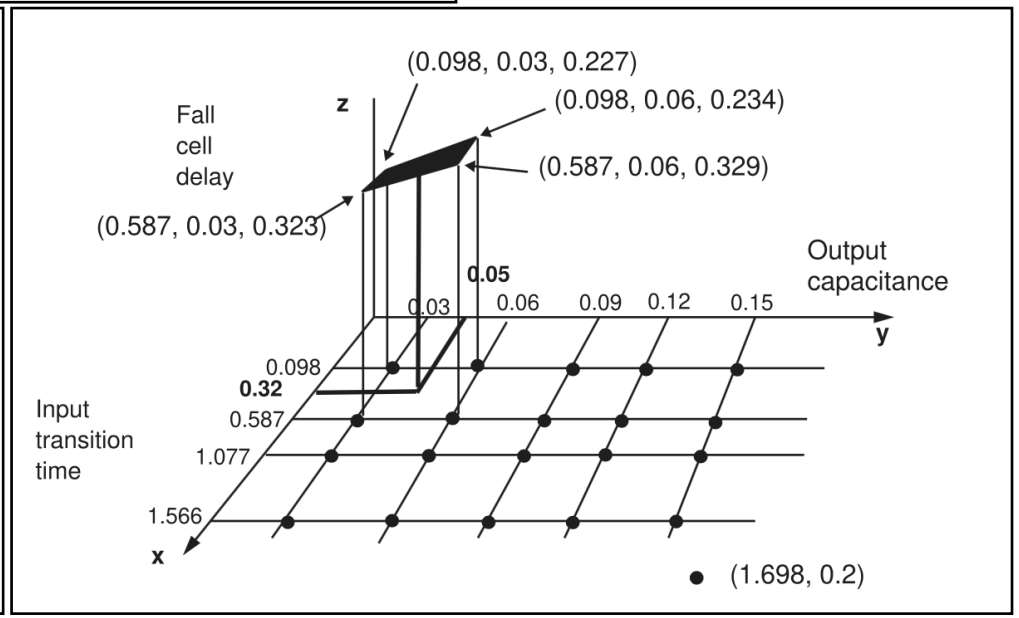
Input transition time is determined by evaluation of the transition delay at the previous gate, U0. Because the timing sense in U1 is negative unate, use the rise transition table for U0 to determine U1's input transition time. If multiple timing arcs are present at gate U0, the maximum of the rise transition values for those arcs is used as the input transition at U1.

Total output capacitance is calculated by addition of the capacitance introduced by the pins connected to net N1 and the capacitance contributed by the wire itself. For wire delay, wire capacitance is calculated by use of a `wire_load` model.

You determine the approximation for the surface described by the three coordinates (x, y, z) in Figure 2 by solving the A, B, C, and D coefficients of the following equation.
Next, insert the coefficient values into the equation to determine which z-coordinate relates to the fall propagation delay.
You can derive the coefficients A, B, C, and D with common mathematical methods, such as Gaussian elimination.

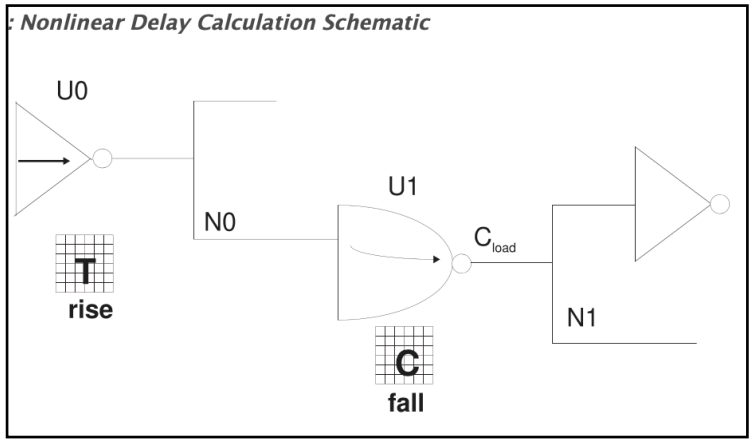
$$Z = A + B*x + C*y + D*x*y$$

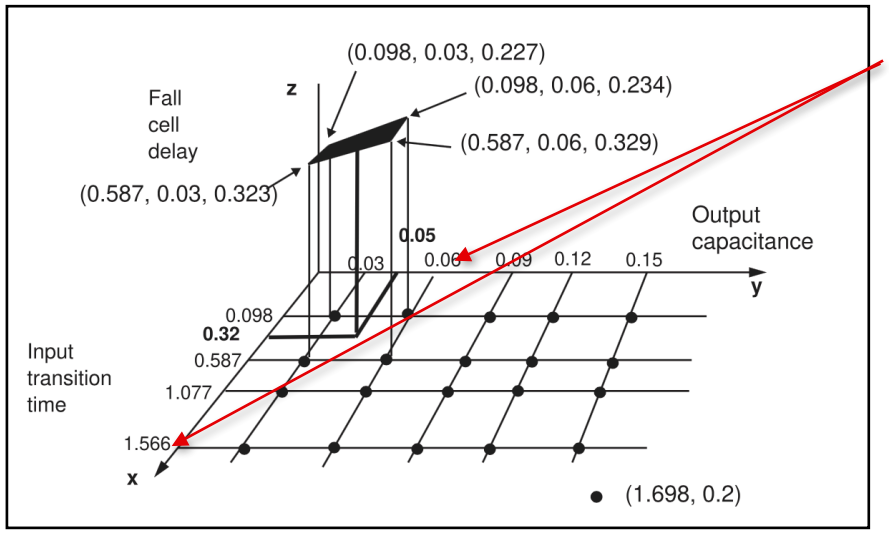




NLDM Delay Calculation (2D Grid)

Input transition time is d





These values are obtained From SPICE simulation.

$$Z = A + B*x + C*y + D*x*y$$

```
0.227 = A + B * 0.098 + C * 0.03 + D * 0.098 * 0.03
0.234 = A + B * 0.098 + C * 0.06 + D * 0.098 * 0.06
0.323 = A + B * 0.587 + C * 0.03 + D * 0.587 * 0.03
0.329 = A + B * 0.587 + C * 0.06 + D * 0.587 * 0.06
```

Insert the coefficient values and the x, y values that equal (0.32, 0.05) to solve for z (fall cell delay):

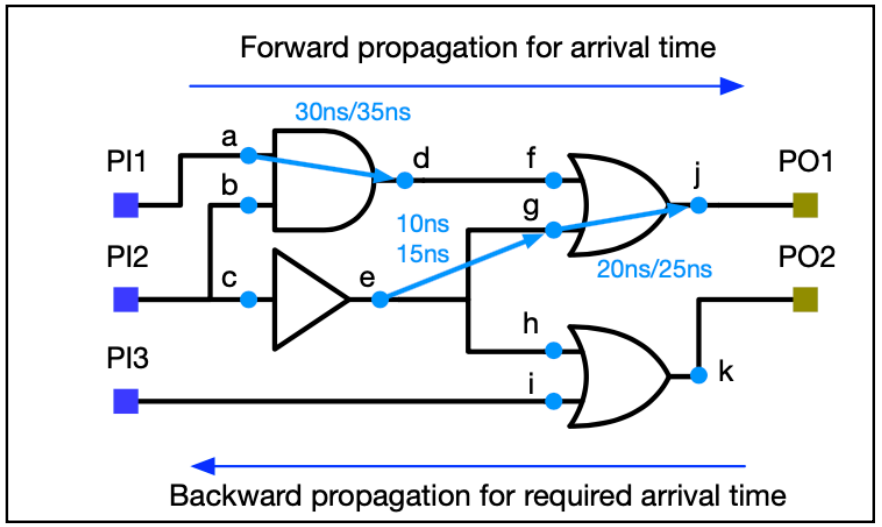
Table 1: Coefficient Values for Fall Cell Delay

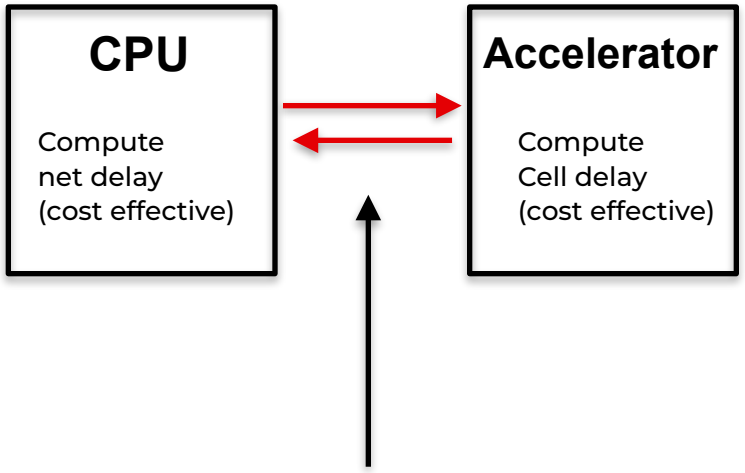
Coefficient	Value
A	0.2006
B	0.1983
C	0.2399
D	0.0677

```
0.275 = 0.2006 + 0.1983 * 0.32 + 0.2399 * 0.05 + 0.0677 *
0.32 * 0.05
```

Parallel Static Timing Analysis Engines

- Computing a STA graph can involve millions or even billions of nodes and edges. The resulting task graph in terms of encapsulated function calls and data dependency is extremely vast and complex
- Existing Work —
- —> Parallel and distributed frameworks to speed up STA. (using CPU's)
- Due to the threading overhead and irregular computational patterns of STA, performance of CPU-based multi-threading usually saturates at around 8–16 threads. To break the performance bottleneck, GPU acceleration for timing analysis is further explored.
- Wang et al proposed acceleration techniques for the look-up table interpolation when computing the cell delays during the timing propagation while the other steps like net delay and levelization are still on CPU. While they demonstrated 6.2X speedup on kernel computation time, the entire propagation runtime becomes 0.9X slower than CPU due to the data transfer overhead.





RunTime distribution of OpenTimer

- **RC Delay Calculation —** They've observed that a significant portion (48%) taken by RC timing, including constructing RC trees and updating parameters for computing slew and delay across nets. Updating RC timing has been always the major bottleneck in analyzing large designs due to large SPEF data needed to process.
- Another large chunk (42%) goes to the construction of the task dependency graph to carry out the timing propagation. Since OpenTimer models each pin as a task, the resulting task graph is linearly proportionally to the size of the STA graph. Constructing a large task graph requires additional layer of data structure to represent tasks and dependencies. Also, only one thread can touch this process at a time, which becomes very time-consuming for large designs.

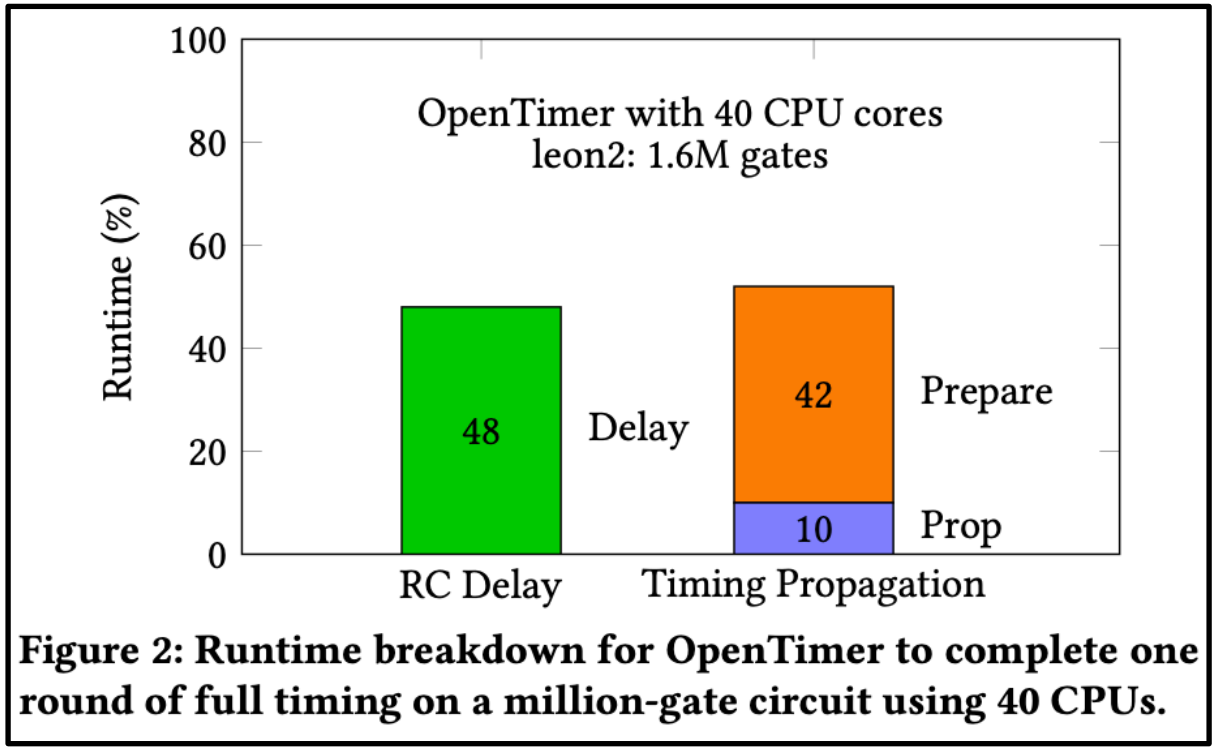
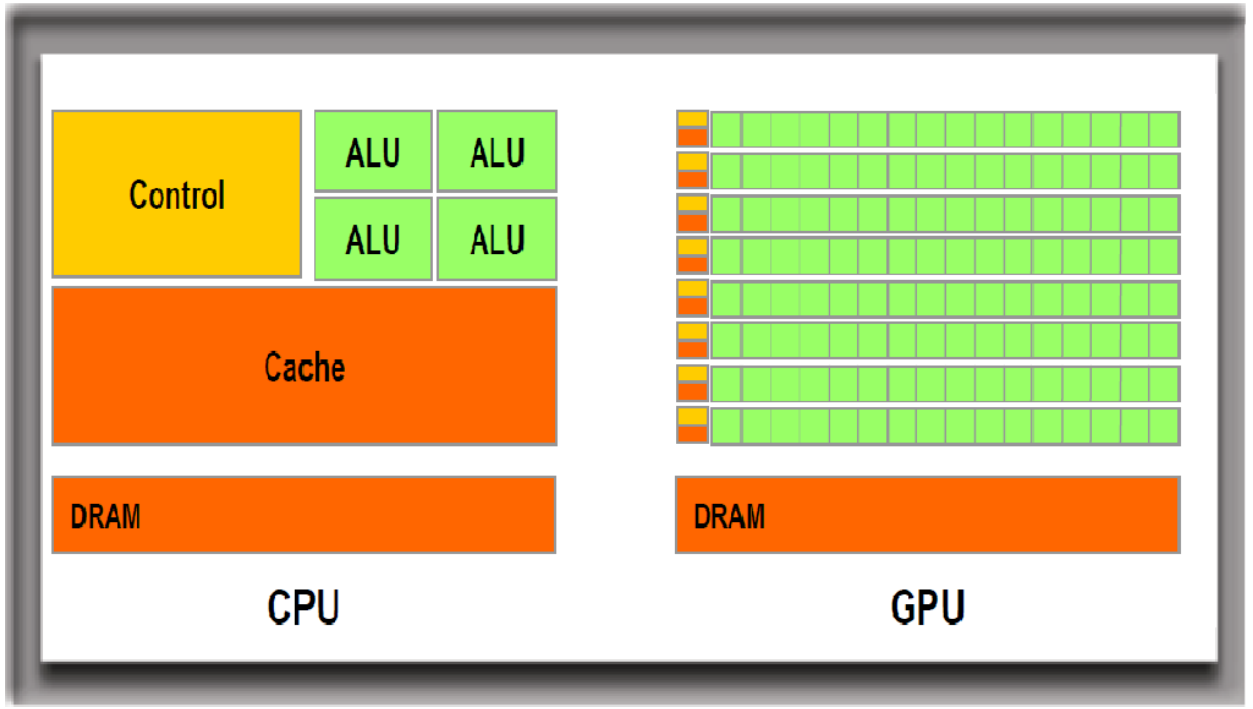


Figure 2: Runtime breakdown for OpenTimer to complete one round of full timing on a million-gate circuit using 40 CPUs.

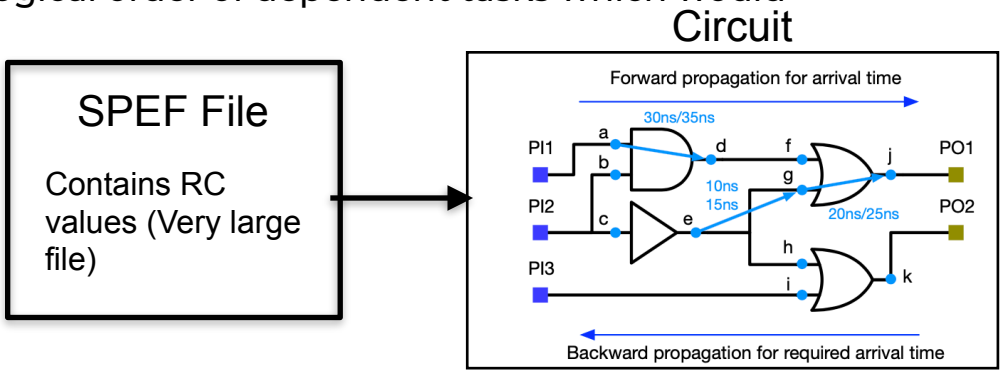
GPU-Accelerated STA

- In general, **CPUs are adopted as the host to manage and schedule all the computation tasks due to its general purpose and powerful control blocks.**
- **Data-intensive computational tasks are offloaded to GPUs for acceleration.** Unlike a CPU, which has a few large “cores” with high performance, a GPU consists of streaming multiprocessors which contain thousands of less powerful small “cores”. It tries to achieve high throughput with massive parallelization at low threading overhead.



Challenges of GPU-Accelerated STA

- The performance characteristics of GPU have the potential to accelerate both delay computation and timing propagation in STA.
- For example, computing the RC timing of each net is independent of each other as we only need to collect parameters for computing slew and delay through the generated RC tree.
- Parasitics data is typically large (gigabytes of SPEF files (Std. Parasitic Exchange Format)) and the computation is data-driven.
- In addition, we may leverage the power of GPU to compute the topological order of dependent tasks which would otherwise be implemented in a single- threaded graph traversal.



- **KEY CHALLENGES —**
- **1. Frequent Memory Access —**

Despite the independent RC delay computation of each net, it requires to access gigabytes memory to complete the computation on million-gate designs, because each net corresponds to an RC tree with the parasitics under different conditions (Early/Late, Rise/Fall)

- **2. Irregular Computation Patterns —**

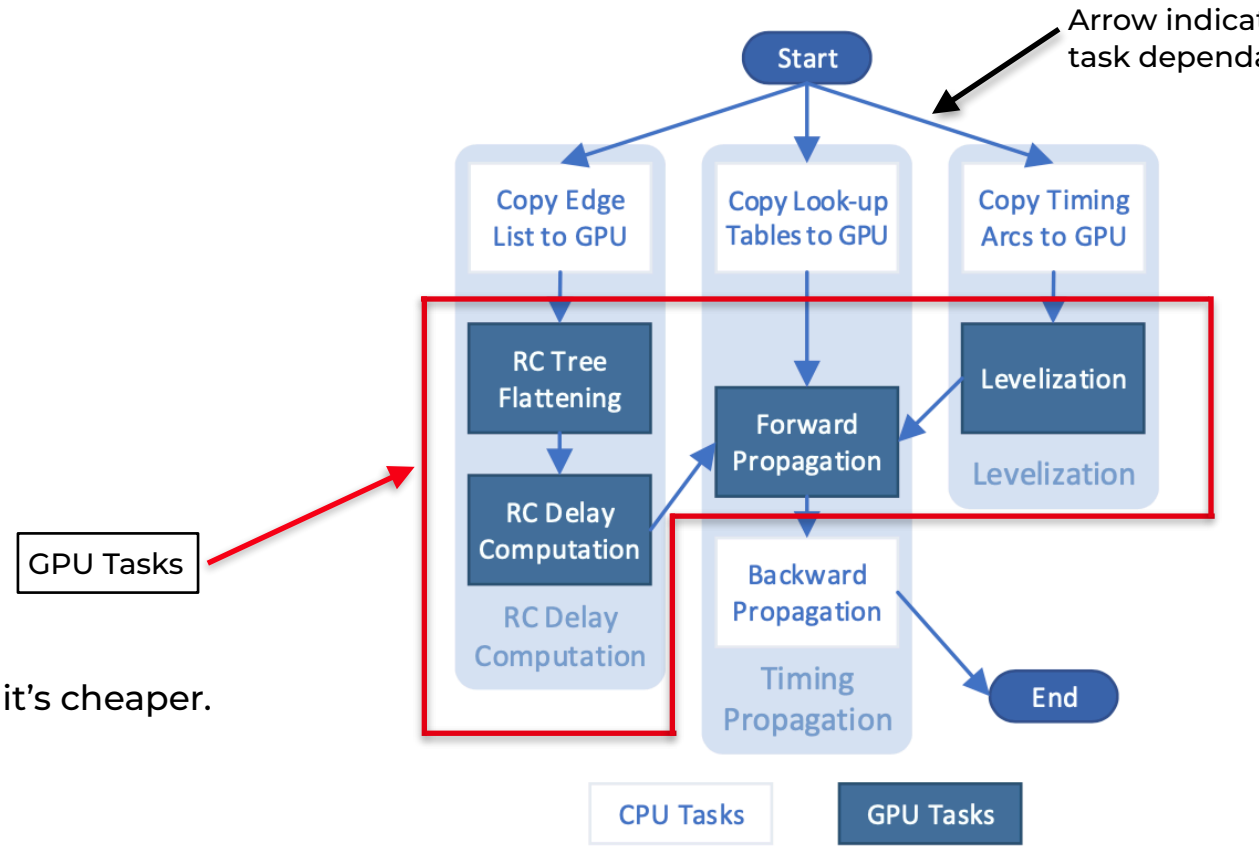
A many STA tasks involve irregular computational patterns, including graph traversal, dynamic data structures, and recursive procedures. Both challenges are associated with each other and require very strategic decomposition algorithms to benefit from GPU parallelism.

Algorithm

- The task graph contains three steps —
- Step 1 — Delay Computation (Cell + Net Delay)
- Step 2 — Levelization
- Step 3 — Timing Propagation

- GPU Tasks —
- 1. RC Tree Flattening
- 2. RC Delay Computation
- 3. Forward Propagation
- 4. Levelization

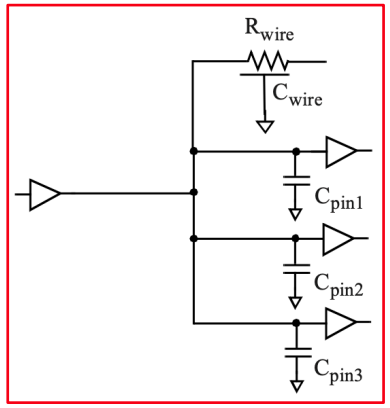
- Note — Backward propagation is left to CPU because it's cheaper.



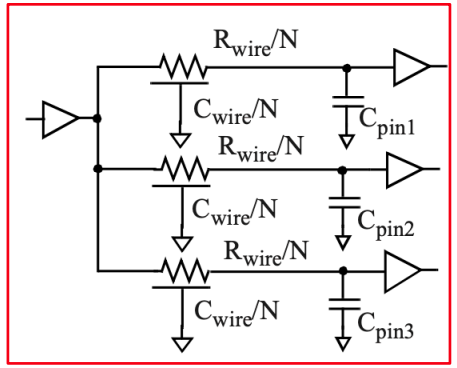
Task-Flow of the GPU Accelerated STA Engine

RC Tree Representations

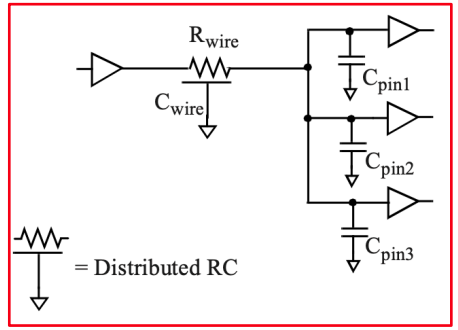
- Two nets with the same total resistance and capacitance, but different RC tree topologies, can have different pin-to-pin delays.
- Three different representations of RC Tree —
- **1. Best Case Tree** — In the best-case tree, it is assumed that the destination (load) pin is physically adjacent to the driver. Thus, none of the wire resistance is in the path to the destination pin. All of the wire capacitance and the pin capacitances from other fanout pins still act as load on the driver pin.
- **2. Balanced Tree** — In this scenario, it is assumed that each destination pin is on a separate portion of the interconnect wire.
- **3. Worst Case Tree** — In this scenario, it is assumed that all the destination pins are together at the far end of the wire. Thus each destination pin sees the total wire resistance and the total wire capacitance.



Best Case Tree



Balanced Tree



Worst Case Tree

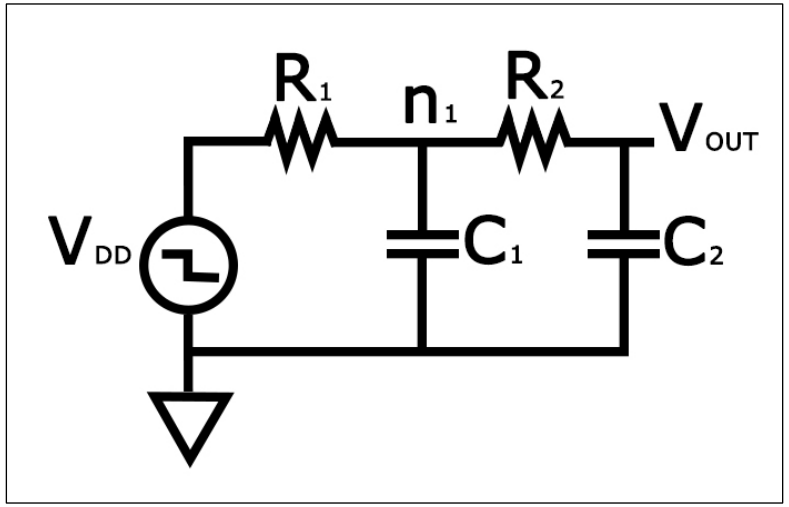
RC Delay Computation —Theory

. Elmore Delay of the RC ladder is given by the following formula

$$t_{pd} = \sum_i R_{is} C_i$$

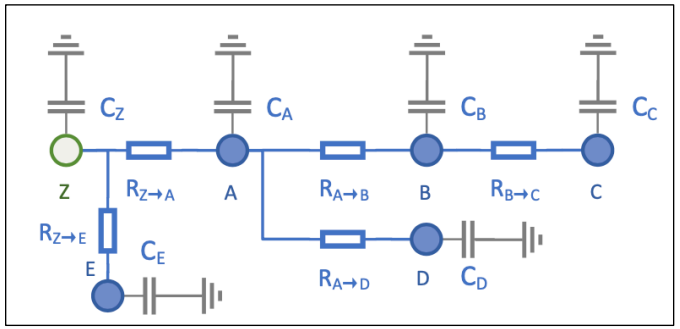
R_{is}= *sum of resistance from source to node i*

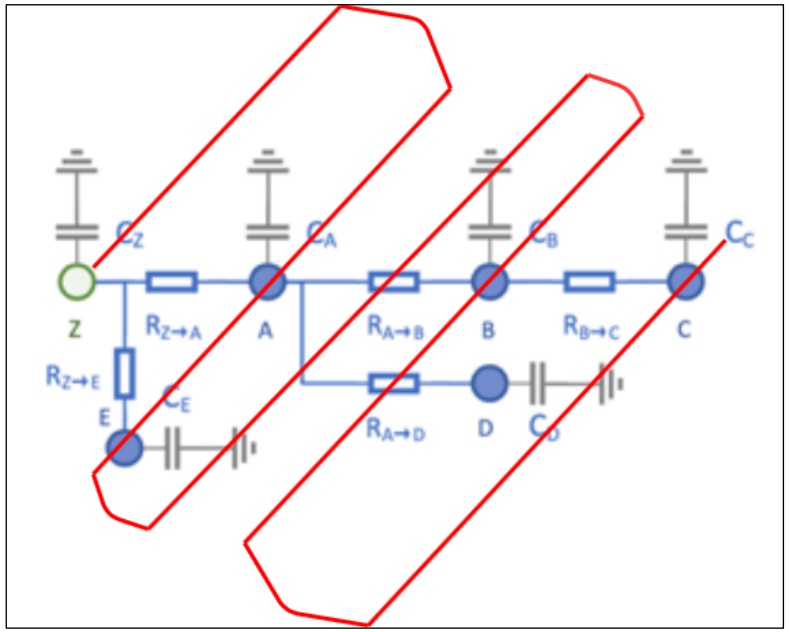
t_pd = R1C1+(R1+ R2)C2

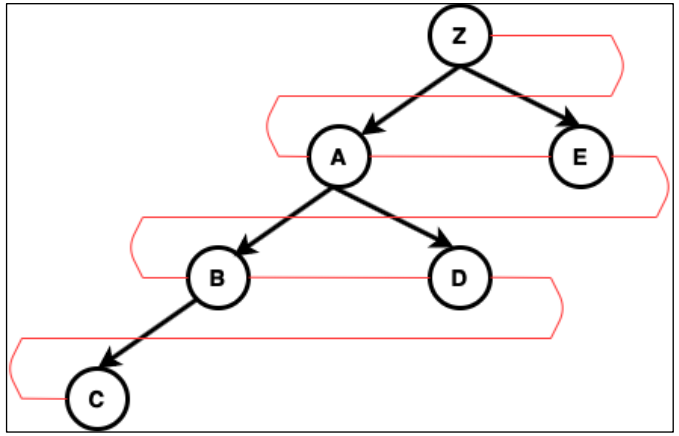


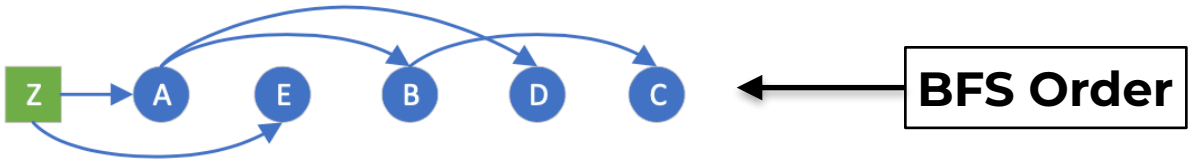
RC Delay Computation

- RC delay computation accounts for the majority of the runtime in most cases.
- The goal is to compute the delay and impulse between the root (Port) and each output pins (Taps).





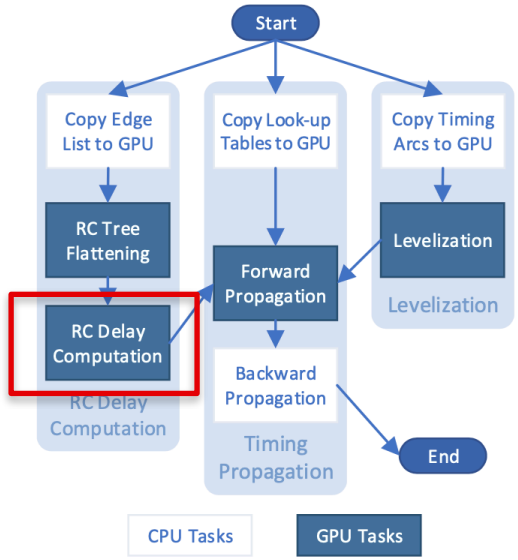




RC Delay Computation Contd.

- How RC delay is computed in the paper —
- A common approach is to compute each parameter through
- multiple depth-first search (DFS) traversals of a RC tree.

- A DFS is not GPU-friendly due to recursive irregularity.
- Obtaining orders of traversed nodes and edges in a RC
- network require non-trivial memory access patterns through
- recursion, which are generally discouraged for
- GPU programming.



CPU Implementation [4]. A common way to implement the RC delay computation is *dynamic programming (DP)*. This algorithm consists of four steps:

(1) Compute the load (i.e. the lumped capacitance) of each node u, denoted as $load_u$.

$$\begin{aligned} load_A &= C_A + C_B + C_C + C_D \\ &= C_A + load_B + load_D. \end{aligned} \tag{1}$$

(2) Compute the delay between *Port* and u, denoted as $delay_u$.

$$delay_u = \sum_{v \in nodes} C_v R_{Port \to \mathrm{LCA}(u,v)}, \tag{2}$$

where LCA denotes lowest common ancestor. [1]

$$\begin{aligned} delay_B =& R_{Z\to A} C_A + R_{Z\to A} C_D + \\ & (R_{Z\to A} + R_{A\to B}) C_B + \\ & (R_{Z\to A} + R_{A\to B}) C_C \\ =& delay_A + R_{A\to B} load_B. \end{aligned} \tag{3}$$

(3) Compute the sum of the product of capacitance and delay in subtrees of u, denoted as $ldelay_u$, similar to step 1.

(4) Compute the beta and impulse value between *Port* and u, using the *ldelay* of each node, similar to step 2.

RC Tree Flattening (BFS Order) Why?

- Tree Flattening Meaning — A “flattening” of a tree is **merely a list resulting from a traversal; your data structure is no longer nested, but flat instead**. To flatten a tree, begin with an empty linked list. Then traverse the tree in the order of your choosing, appending each visited node to the linked list.
- We precompute a BFS order for each tree. Based on this order, every parent
- appears before all its children. If there is an edge u → v, then u appears before v.
-
- This BFS order is a GPU-efficient representation of a RC tree. We only need to
- traverse the nodes through the ordered sequence, either forward or backward,
- according to the direction of the DP step.

- Two Steps in the Algorithm —
- 1. Compute the distances of each node to the root.
- 2. Sort nodes according to their distance to the root.

GPU Tasks —

1. RC Tree Flattening
2. RC Delay Computation
3. Forward Propagation
4. Levelization

Note — These operations are performed on the GPU

Algorithm 1: Flatten RC Trees

Input: N as #nets, (M, E) as (#nodes, #edges) in all nets
Input: $roots[0..N-1]$, the index of root of each net
Input: $edges[0..E-1]$, the undirected edges $\{(a, b)\}$
Input: $nodestart[0..N]$, the offsets of each net in arrays of nodes, with $nodestart[N] = M$
Input: $edgestart[0..N]$, the offsets of each net in arrays of edges, with $edgestart[N] = E$
Input: $distances[0..M] = \infty$, $counts[0..M] = 0$
Output: $order[0..M-1]$, nodes in BFS order for each net

```
/* Process one net w/ blockDim.x threads                    */
1  netID = blockIdx.x;                          ▷ gridDim.x = #nets
2  threadID = threadIdx.x;                      ▷ blockDim.x = 64
3  nst = nodestart[netID];                      ▷ node offset start
4  nend = nodestart[netID + 1];                 ▷ node offset end
5  est = edgestart[netID];                      ▷ edge offset start
6  eend = edgestart[netID + 1];                 ▷ edge offset end
7  distances[nst + roots[netID]] = 0;
8  for d = 0, 1, 2, ..., (nend − nst) do
9      for i = est + threadID to eend step blockDim.x do
10         (a, b) = edgelist[i];
11         if distances[a] == d and distances[b] >d + 1 then
12             distances[b] = d + 1;
13             atomicAdd(counts[d], 1);
14         end
15         else if distances[b] == d and distances[a] >d + 1 then
16             distances[a] = d + 1;
17             atomicAdd(counts[d], 1);
18         end
19     end
20     __syncthreads();                         ▷ Sync threads within a block
21     break when counts[d] == 0;
22 end
23 countingSort(distances, counts, order, threadID);
```

GPU Tasks —
1. RC Tree Flattening
2. RC Delay Computation
3. Forward Propagation
4. Levelization

RC Delay Computation Contd.

CPU Implementation [4]. A common way to implement the RC delay computation is *dynamic programming (DP)*. This algorithm consists of four steps:

(1) Compute the load (i.e. the lumped capacitance) of each node u, denoted as $load_u$.

$$\begin{aligned} load_A &= C_A + C_B + C_C + C_D \\ &= C_A + load_B + load_D. \end{aligned} \tag{1}$$

(2) Compute the delay between *Port* and u, denoted as $delay_u$.

$$delay_u = \sum_{v \in nodes} C_v R_{Port \to \mathrm{LCA}(u,v)}, \tag{2}$$

where LCA denotes lowest common ancestor. [1]

$$\begin{aligned} delay_B &= R_{Z\to A} C_A + R_{Z\to A} C_D + \\ &\quad (R_{Z\to A} + R_{A\to B}) C_B + \\ &\quad (R_{Z\to A} + R_{A\to B}) C_C \\ &= delay_A + R_{A\to B} load_B. \end{aligned} \tag{3}$$

(3) Compute the sum of the product of capacitance and delay in subtrees of u, denoted as $ldelay_u$, similar to step 1.

(4) Compute the beta and impulse value between *Port* and u, using the *ldelay* of each node, similar to step 2.

Algorithm 2: Compute RC Delay

Input: N as #nets, M as #nodes in all nets
Input: $start[0..N]$, the offsets of each net in arrays of nodes
Input: $parent[0..M-1]$, the index of parent of every nodes
Input: $pres[0..M-1]$, the resistance between nodes and their parent
Input: $cap[0..4M-1]$, the capacitance of nodes, each in 4 different combinations
Output: $load[0..4M-1]$, $delay[0..4M-1]$, $impulse[0..4M-1]$: arrays of results of load, delay and impulse, respectively

```
1  netID = blockIdx.x × blockDim.x + threadIdx.x;
2  condID = threadIdx.y;
3  if netID ≥ N then return;
4  offsetL = start[netID];                    ▷ node offset start
5  offsetR = start[netID + 1];                ▷ node offset end
6  Initialize load, delay, ldelay to zero;
7  Initialize β = 0 as an auxiliary array;
8  for i = offsetR − 1 down to offsetL do
9      load[4i + condID] += cap[4i + condID];
10     load[4parent[i] + condID] += load[4i + condID];
11 end
12 for i = offsetL + 1 to offsetR − 1 do
13     t = load[4i + condID] × pres[i];
14     delay[4i + condID] = delay[4parent[i] + condID] + t;
15 end
16 for i = offsetR − 1 down to offsetL do
17     ldelay[4i + condID] +=
         cap[4i + condID] × delay[4i + condID];
18     ldelay[4parent[i] + condID] += load[4i + condID];
19 end
20 for i = offsetL + 1 to offsetR − 1 do
21     t′ = ldelay[4i + condID] × pres[i];
22     β[4i + condID] = β[4parent[i] + condID] + t′;
23     impulse[4i + condID] =
         2β[4i + condID] − delay[4i + condID]^2;
24 end
```

Theory behind RC delay calculation

- We launch the kernel per net under each Early/Late and Rise/Fall condition.
- A total of 4N threads are launched, where N is the number of nets. Initially, the netID and condID is computed in line 1-3.
- We compute the offsets of the data for the net in the arrays in line 4-5 and initialize the output arrays with zeros in line 6-7.
- Then, we traverse and update the values of load (line 8-11), delay (line 12-15), ldelay (line 16-19), beta and impulse (line 20-24).
- We store the parent index of each node in array parent. This provides a workload-balanced
- parent-child representation on GPU, while preserving our ability to perform DP updates.

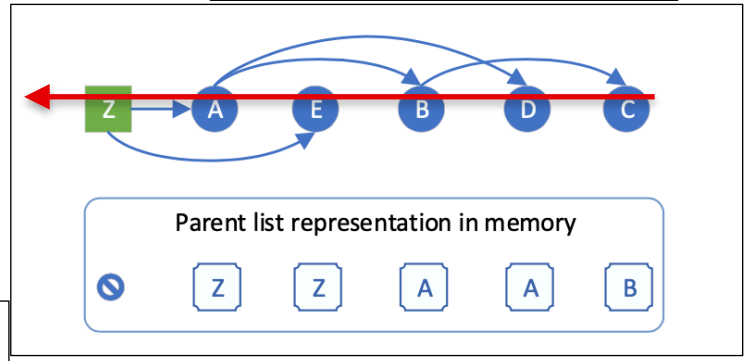
Capacitive Load Calculation & Delay Calc.

the recursive equation for *load* is

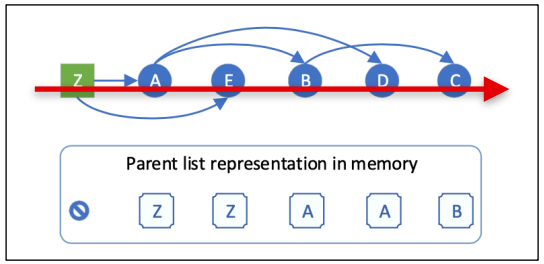
$$load_u = cap_u + \sum_{v \in \{\text{children of } u\}} load_v,$$

Running sum

Backward traversal of node sequence

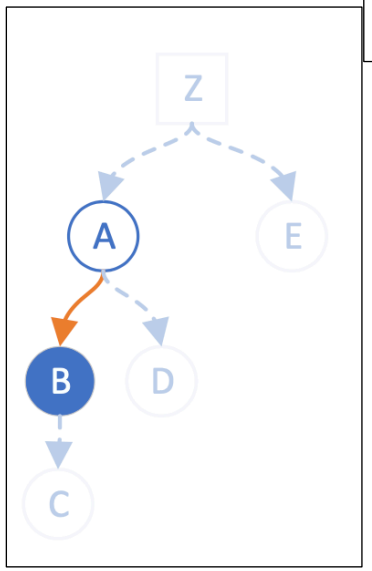


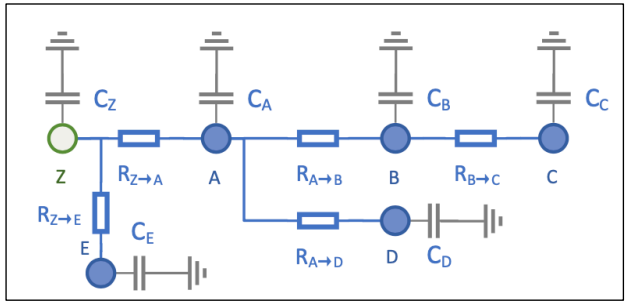
$$delay_v = delay_u + pres_v \times load_v$$



$$\begin{aligned} delay_B =& R_{Z\to A}C_A + R_{Z\to A}C_D + \\ & (R_{Z\to A} + R_{A\to B})C_B + \\ & (R_{Z\to A} + R_{A\to B})C_C \\ =& delay_A + R_{A\to B} load_B. \end{aligned}$$

Forward traversal of node sequence





Eg. Load @ Node A = Cap @ Node A + Load (B+D+C)

GPU Memory Access Latency Opt.

- We optimize the global memory access latency when the number of nets is large and we have to update the RC at different conditions using different threads.
- Data stored in GPU's global memory will incur some latency when accessed.
- In practice, GPU will identify threads that request memory access to adjacent addresses, and coalesce these requests.
- Thus, we interleave the memory of the four threads performing independently on each of the
- four Early/Late and Rise/Fall conditions, instead of storing them separately
-

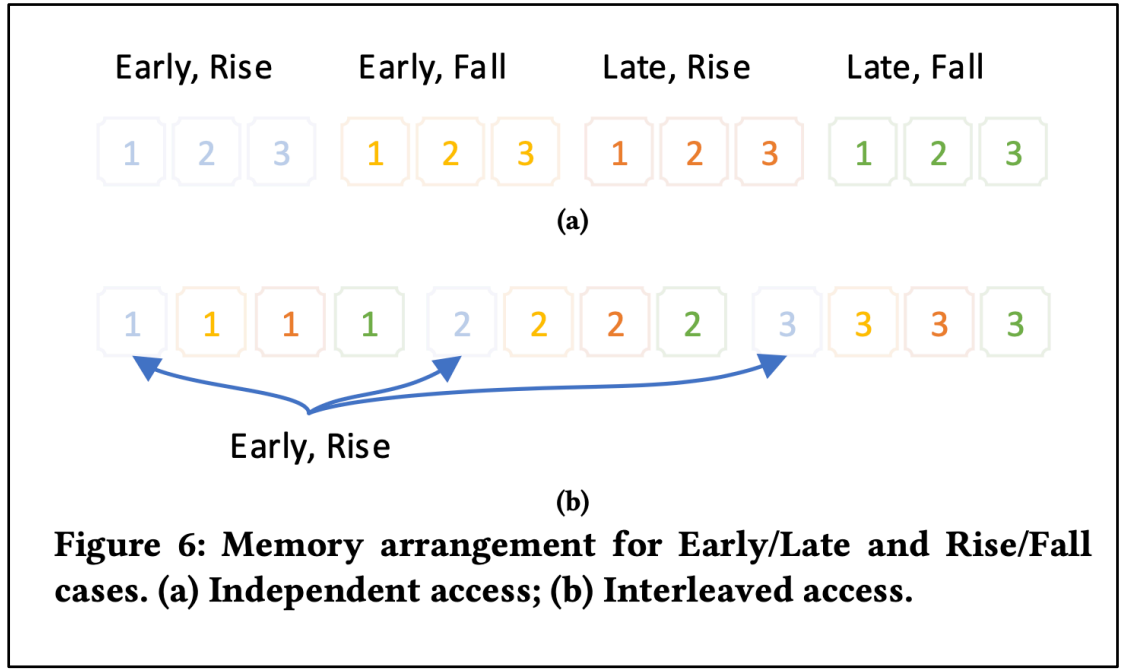
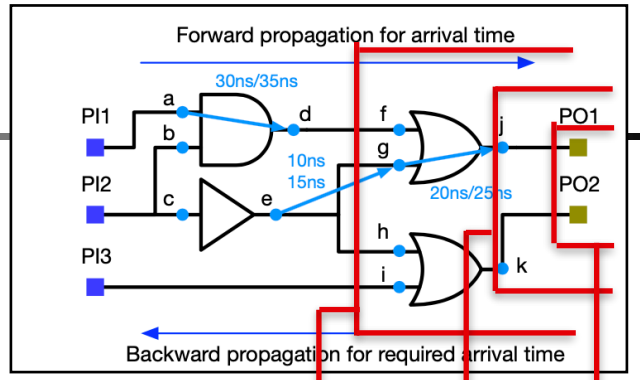


Figure 6: Memory arrangement for Early/Late and Rise/Fall cases. (a) Independent access; (b) Interleaved access.

Levelization

- Levelization is a preparation step for timing propagation.
- **It builds level- by-level dependencies for propagation tasks and accounts for nearly 40% of the runtime**
- A root cause is its single-threaded pattern. Existing timers, including industrial tools construct a level list
- data structure for all logic levels through a single-threaded BFS or DFS.
- Tasks within the same level can run in parallel; Lower- level tasks must not run after tasks at higher levels.
- This is time consuming. The authors use GPU to accelerate this procedure.
-

Levelization

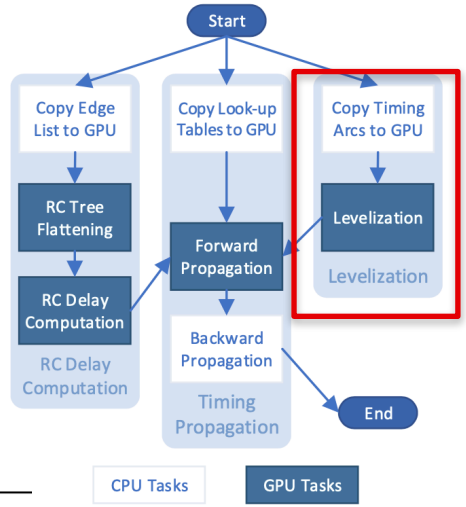


- The procedure of our GPU-accelerated levelization is shown as Algorithm 3.
- **The key idea is to maintain a set of nodes in the present level, called frontiers, denoted as F .**
- The initial frontiers are nodes that do not have input edges (line 1).
- The algorithm loops through line 3-6 until all nodes are discovered.
- At each iteration, we invoke a GPU kernel function **advanceFrontier** to discover the next frontiers in parallel based on the current ones.

Algorithm 3: Levelize

Input: the set of nodes *nodes*
Data: the adjacency list *out*, the current in-degree *in*
Output: a level list of nodes

1 $F \leftarrow \{f \in nodes : in_f = 0\}$;
2 **while** F *is not empty* **do**
3 output F;
4 $F' \leftarrow \{\}$;
5 Call advanceFrontier on F and get F';
6 $F \leftarrow F'$;
7 **end**



Algorithm 4: Advance Frontier

Input: the old frontier F
Data: the adjacency list *out*, in-degree array *in*
Output: the new frontier F'

1 *nodeID* ← **blockIdx**.x × **blockDim**.x + **threadIdx**.x;
2 **if** *nodeID* ≥ size(F) **then return**;
3 **for** v *in out*[*nodeID*] **do**
4 *oldvalue* ← atomicAdd(*in*[v], -1);
5 **if** *oldvalue* = 1 **then**
6 Add v to F';
7 **end**
8 **end**
9 **return** G;

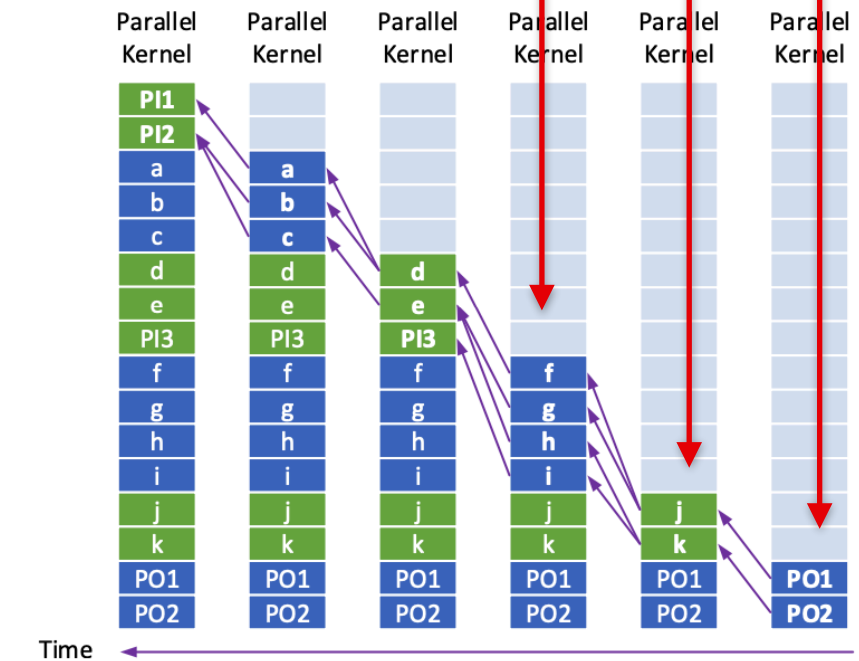
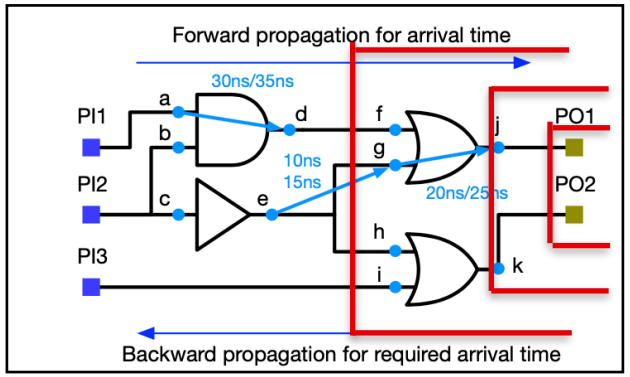


Figure 7: Levelization of the timing graph in Figure 1 using GPU. Nodes in bold are frontiers at the corresponding iteration. At each iteration, we obtain one level of nodes.

Advance Frontier

- Algorithm 4 shows the pseudocode for advanceFrontier.
- This algorithm works on current frontiers and each thread processes one frontier.
- We enumerate all output edges of each frontier (line 3-8).
- For each output edge with destination v, we decrease the in-degree of v by one.
- If the in-degree of v becomes zero afterwards, we add v to the set of next frontiers.



Algorithm 4: Advance Frontier

Input: the old frontier F
Data: the adjacency list *out*, in-degree array *in*
Output: the new frontier F'

```
1 nodeID ← blockIdx.x × blockDim.x + threadIdx.x;
2 if nodeID ≥ size(F) then return;
3 for v in out[nodeID] do
4 |   oldvalue ← atomicAdd(in[v], -1);
5 |   if oldvalue = 1 then
6 |   |   Add v to F';
7 |   end
8 end
9 return G;
```

LUT Interpolation

. Working explained in slide 9. LUT Interpolation on GPU gives slightly better
. performance.

Algorithm 5: LUT Interpolation

```
   /* Input: line (x1, y1)--(x2, y2)                                  */
   /* Input: the x value queried                                      */
1  Function interpolate(x1, x2, y1, y2, x):
2      if x1 = x2 then return y1;
3      else return d1 + (d2 − d1) (x−x1)/(x2−x1);
4  end
   /* Input: n × m look-up table                                      */
   /* Input: the point queried (x, y)                                 */
5  Function lut_lookup(n, m, X, Y, mat, x, y):
6      i′ ← 0;
7      i ← min(1, n − 1);
8      while i + 1 < n and X[i] ≤ x do
9          i′ ← i;
10         i ← i + 1;
11     end
12     j′ ← 0;
13     j ← min(1, m − 1);
14     while j + 1 < m and Y[j] ≤ y do
15         j′ ← j;
16         j ← j + 1;
17     end
18     r_i′ ←interpolate(Y[j′], Y[j], mat[i′, j′], mat[i′, j]);
19     r_i ←interpolate(Y[j′], Y[j], mat[i, j′], mat[i, j]);
20     r ←interpolate(X[i′], X[i], r_i′, r_i);
21     return r;
22 end
```

Experimental Results

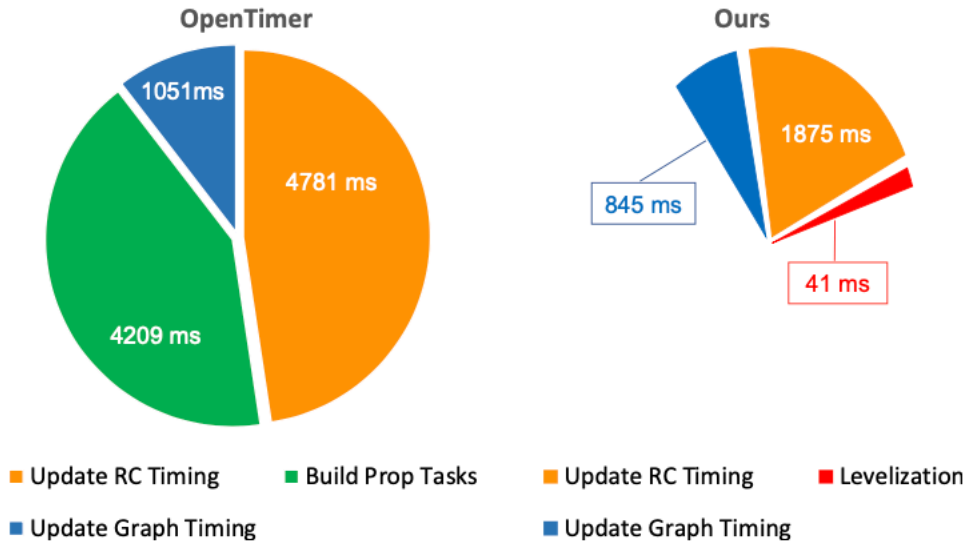


Figure 9: Runtime breakdown of the circuit leon2 (21M nodes).

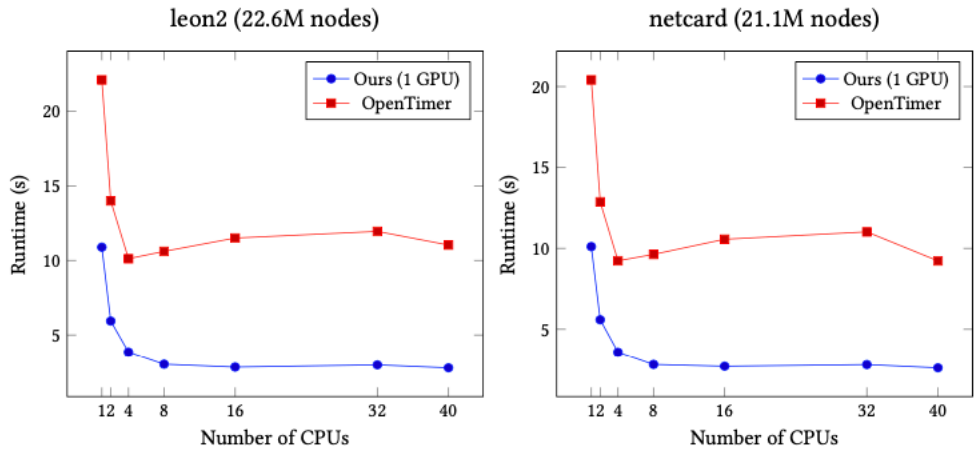


Figure 10: Runtime values at different numbers of CPUs. Our runtime under 1 CPU and 1 GPU is close to OpenTimer of 40 CPUs.

Incremental Timing

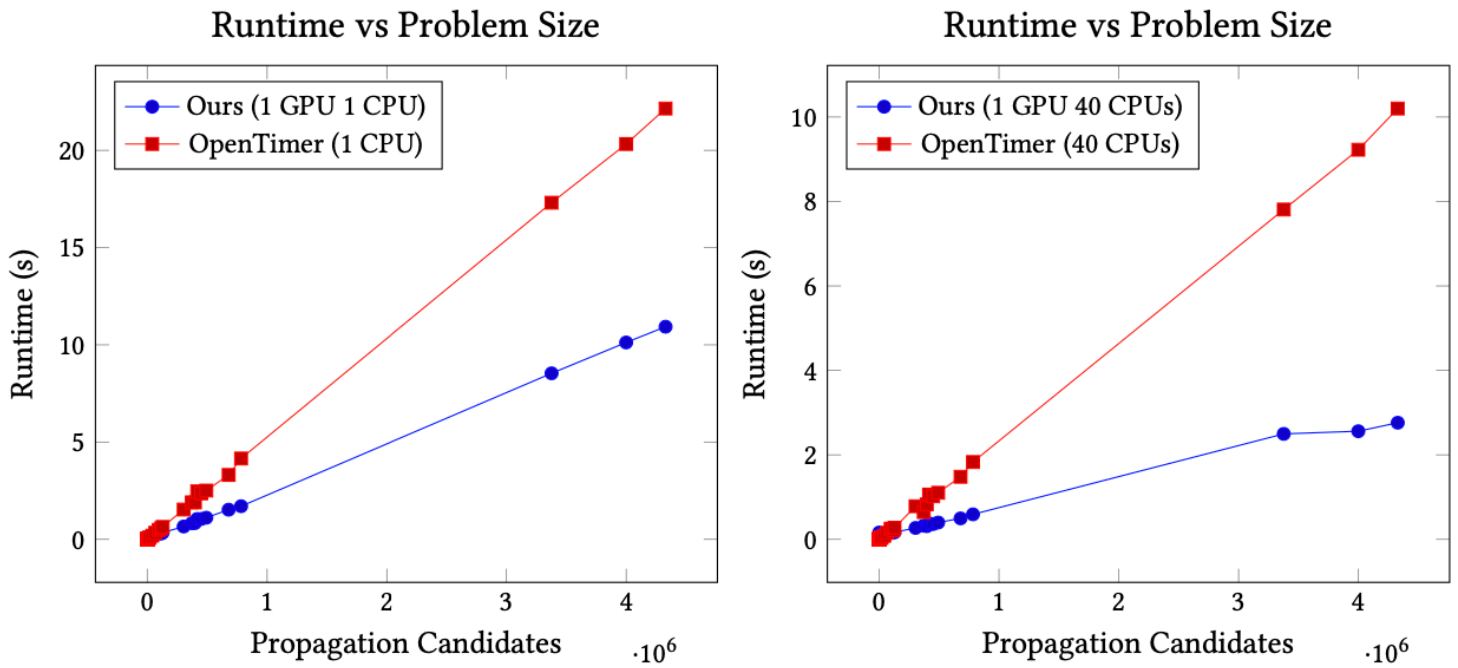


Figure 11: Runtime values at different problem sizes. Beyond about 60K propagation candidates, our runtime is always faster than OpenTimer at any CPU numbers.

Comparison

Table 1: Performance comparison between OpenTimer (40 CPUs) and our GPU-accelerated Implementation (1 GPU) to complete one iteration of full timing on large circuit designs (>10K gates) of TAU 2015 contest benchmarks

Benchmark	# PIs	# POs	# Gates	# Nets	# Pins	# Nodes	# Edges	OpenTimer Runtime (40 CPUs)	Our Runtime (40 CPUs 1 GPU)	
									Runtime	Speed-up
aes_core	260	129	22938	23199	66751	413588	453508	156 ms	138 ms	1.13×
vga_lcd	85	99	139529	139635	397809	1966411	2185601	829 ms	311 ms	2.67×
vga_lcd_iccad	85	99	259067	259152	679258	3556285	3860916	1480 ms	496 ms	2.98×
b19	22	25	255278	255300	782914	4423074	4961058	1831 ms	585 ms	3.13×
cordic	34	64	45359	45393	127993	7464477	820763	274 ms	167 ms	1.64×
des_perf	234	140	138878	139112	371587	2128130	2314576	832 ms	325 ms	2.56×
edit_dist	2562	12	147650	150212	416609	2638639	2870985	1059 ms	376 ms	2.86×
fft	1026	1984	38158	39184	116139	646992	718566	241 ms	148 ms	1.63×
leon2	615	85	1616369	1616984	4328255	22600317	24639340	10200 ms	2762 ms	3.69×
leon3mp	254	79	1247725	1247979	3376832	17755954	19408705	7810 ms	2585 ms	3.02×
netcard	1836	10	1496719	1498555	3999174	21121256	23027533	9225 ms	2571 ms	3.60×
mgc_edit_dist	2562	12	161692	164254	450354	2436927	2674934	1021 ms	368 ms	2.77×
mgc_matrix_mult	3202	1600	171282	174484	492568	2713241	2994343	1138 ms	377 ms	3.02×
tip_master	778	857	37715	38493	95524	533690	570154	163 ms	143 ms	1.14×

 # PIs: number of primary inputs **# POs**: number of primary outputs **# Gates**: number of gates **# Nets**: number of nets
 # Pins: number of pins **# Nodes**: number of nodes in the STA graph **# Edges**: number of edges in the STA graph

Conclusion

- GPU accelerated STA Engines give a better run time than conventionally designed STA Engines.